REVISTA Universidad EAFIT
Vol. 42. No. 141. 2006. pp. 60-76

Una visión de la enseñanza de la Ingeniería de Software como apoyo al mejoramiento de las empresas de software

Raquel Anaya
Ph. D. en Informática. Profesora del Departamento de Informática y Sstemas de la Universidad EAFIT y Directora del Grupo de Investigación en Ingeniería de Software de la misma universidad.
ranaya@eafit.edu.co

Recepción: 01 de junio de 2005 | Aceptación: 23 de febrero de 2006

Resumen

El desarrollo de software es una actividad compleja que requiere la integración de factores técnicos, gerenciales y organizacionales. Consecuentemente, es un reto para la academia preparar estudiantes orientados a mejorar la práctica del desarrollo en las organizaciones de software. Los principios y estrategias propuestos en este trabajo, parten de dos requerimientos básicos: a) Los profesores deben tener una vista unificada acerca del cuerpo de conocimiento que soporta esta área; b) La academia debe tener un percepción de la realidad de esta práctica en las organizaciones de software y los problemas que dichas organizaciones enfrentan, debido a la falta de aplicación de buenas prácticas de ingeniería de software. Las estrategias definidas están encaminadas a formar las competencias requeridas de tal forma que los futuros desarrolladores estén convencidos que las prácticas de ingeniería de software son claves para el desarrollo exitoso de software.

Palabras Clave

Ingeniería de software
Enseñanza de la ingeniería de software
Estado de las empresas de software
La industria de software
Aprendizaje basado en competencias
Principios pedagógicos de enseñanza de la ingeniería de software

A vision of Software Engineering teaching as support to the improvement of software companies

Abstract

Software development is a complex activity that requires the integration of technical, managerial and organizational factors. Consequently, it is a challenge for the academic training students focused on the improvement of the software engineering practice in a business context. The principles and strategies exposed in this work, are based on two basic requirements: a) Teachers should have a unified view about the knowledge body that support this area; b) The academy should know the reality of the software organizations and the problems these have; many of these problems exist because good software engineering practices are not being carried out in software development projects. The strategies are oriented to shape the required competitions that will result in software engineering graduates that will be sure that good software engineering practices are central for successful software development.

Palabras Clave

Software Engineering
Teaching of software engineering
Status of software engineering companies
Software industry
Learning based on competition
Pedagogical principles for the teaching of software engineering

Introducción

La rápida evolución de la tecnología informática ha propiciado la definición de nuevos servicios y esquemas de trabajo en las organizaciones, que le permiten a éstas mejorar la calidad de servicio a sus clientes, definir nuevos servicios, conquistar nuevos mercados y, en general, ser más competitivas. La operación de la compañía se encuentra cada vez más soportada en sistemas de información intensivos en software los cuales son fundamentales para apoyar el liderazgo estratégico de la organización en el mercado o, por el contrario, propiciar su fracaso. Estas condiciones de contexto generan permanentes demandas a la Ingeniería de software, con una premisa fundamental: la calidad.

El desarrollo de software ha sido considerado tanto un arte como una ciencia (Lawrence, 2002). Como ciencia, esta disciplina se fundamenta en la aplicación de prácticas de ingeniería que permiten estimar, medir y evaluar el proceso de desarrollo, de manera repetitiva y controlada. Como arte, el desarrollo de software requiere producir resultados que reflejen ingenio y habilidad para seleccionar, diseñar y construir el producto software que mejor satisfaga los requerimientos de una organización. Las características naturales del producto como maleabilidad, intangibilidad, subjetividad y naturaleza discreta, unidas a las condiciones del contexto, como evolución de la tecnología informática, relaciones interpersonales complejas e indefinidas y dinámica permanente del negocio, imponen una complejidad particular a esta disciplina que la diferencia de las otras áreas de la ingeniería (Bruegge y Dutoit, 2002; Maibaum, 2004).

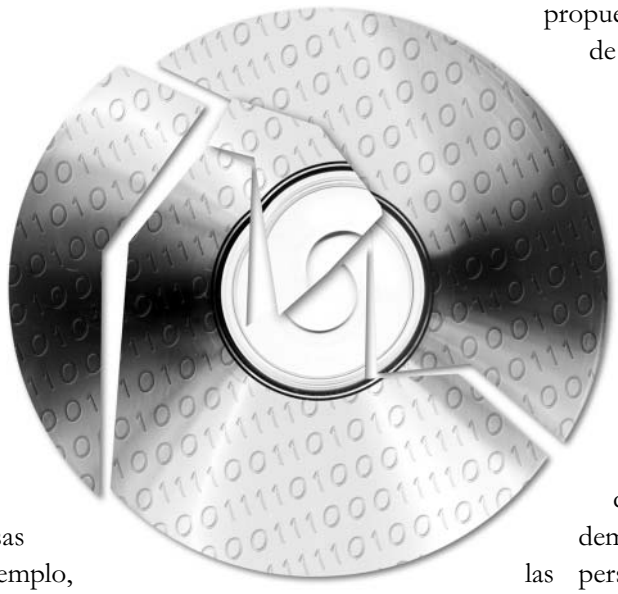
El desarrollo de software se ha convertido en una actividad tan especializada, que ha originado un desplazamiento de la actividad hacia compañías cuya actividad fundamental es el desarrollo de software, permitiendo que las empresas de producción o servicio se dediquen a las actividades orientadas hacia sus objetivos organizacionales. Este fenómeno creó una nueva actividad económica alrededor de la industria del software que hoy se considera promisoria para los países latinoamericanos. Sin embargo, los estudios actuales coinciden en afirmar que las empresas de software latinoamericanas están en desventaja frente a los niveles de competitividad exigidos a nivel internacional (Mayer y Bunge, 2004; Valdes, 2004).

Existe una brecha marcada entre lo que se enseña en el aula de clase y la realidad del desarrollo de

software en las organizaciones. Esta situación ha motivado una reflexión en el mundo académico, acerca de las competencias y habilidades que deben ser desarrolladas en los futuros desarrolladores de software y las estrategias pedagógicas que pueden ser utilizadas de manera que sus experiencias de aprendizaje estén altamente influenciadas por las prácticas, técnicas y modos de trabajo que exige el desarrollo de software de calidad a escala industrial (Stiller y LeBland, 2002; Liu et. al., 2002; Bracken, 2003; Maibaum, 2004).

El objetivo de este trabajo es hacer un primer acercamiento de tipo reflexivo a la enseñanza de la ingeniería de software, considerando el objeto de conocimiento y la realidad de las organizaciones de software. En la sección 1 se analiza el objeto de conocimiento desde la perspectiva de SWEBOK (2004) y los principios que de dicho objeto de aprendizaje se derivan. En la sección 2 se presenta la realidad de las empresas de software y sus problemáticas con respecto a una aplicación no adecuada de dichos principios. Esta situación contrastante entre el objeto de conocimiento y la realidad empresarial, servirá de punto de partida para presentar, en la sección 3, los principios y estrategias en los que se debe soportar la enseñanza de la ingeniería de software de tal manera que se favorezca el desarrollo de competencias que aporten al mejoramiento de las empresas de software. Finalmente se presentan las conclusiones y trabajos que se siguen al respecto.

1. Referente conceptual de la ingeniería de software



El proceso, el producto, el método, las personas y la tecnología son los elementos básicos involucrados en el desarrollo de software, los cuales puede ser interrelacionados de diversas maneras. Pressman, por ejemplo, establece una relación de jerarquía de dichos elementos al cual denomina tecnología multicapa, que parte del enfoque de calidad sobre el cual se soporta el proceso, el método y las herramientas (Pressman, 2002). Un adecuado proceso de formación en la enseñanza de la ingeniería de software debe considerar de manera natural las interrelaciones que se suceden entre dichos elementos que, de acuerdo a Paulish, representan los "factores controlables en la mejora de la calidad del software y el desempeño organizacional" (citado por Pressman, 2006, p. 664).

De otra parte, a través del proyecto SWEBOK (Guide to the Software Engineering Body of Knowledge) (SWEBOK, 2004), la comunidad informática pretende establecer un referente conceptual de la ingeniería de software, considerando los requerimientos de la práctica de desarrollo en la industria. Este proyecto representa un esfuerzo de unificación de organismos como IEEE y Computer Society, quienes, junto con reconocidas instituciones académicas, han venido trabajando desde 1979 para definir un cuerpo unificado de conocimiento de esta disciplina. En su versión 2004, SWEBOK presenta, una estructura conformada por 11 áreas de conocimiento KA (Knowledge Area) que integran las perspectivas teóricas y prácticas propias de la ingeniería de software (Figuras 1a y 1b) .

Es interesante analizar la relación que existe entre los elementos del proceso software propuestos por Paulish y las áreas de conocimiento definidas en SWEBOK: mientras que las KA´s presentadas en la Figura 1a (KA1 a KA5), hacen referencia a disciplinas de ingeniería con un propósito particular dentro del ciclo de vida del proyecto, las presentadas en la cuadro Figura 1b (KA6 – KA10), hacer referencia a disciplinas de apoyo a las demás. El proceso, el producto, las personas, la tecnología son

elementos que están siempre presentes en las diferentes disciplinas con algunas variantes particulares justificadas por el desarrollo histórico del área de conocimiento. La Tabla 1 presenta una visión sintetizada del estado actual del área de conocimiento enfatizando los principios conceptuales que de ahí se derivan.

Figura 1a. Estructura del cuerpo de conocimiento SWEBOK – Parte 1

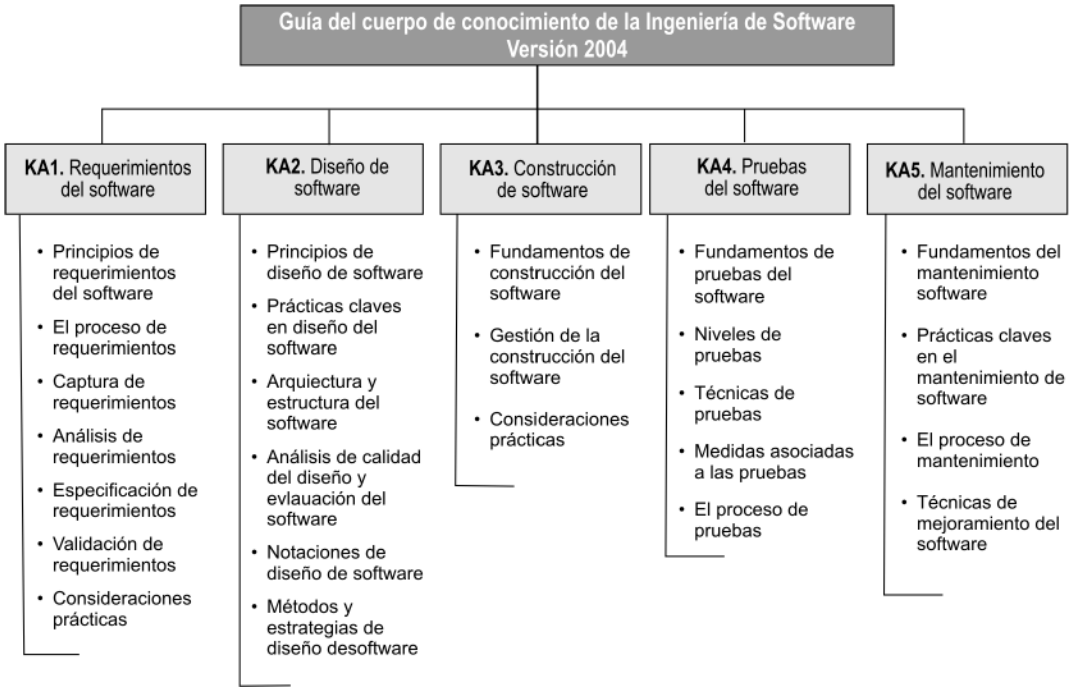


Figura 1b. Estructura del cuerpo de conocimiento SWEBOK – Parte 2

Tabla 1. Visión sintetizada del estado actual del área de conocimiento enfatizando los principios conceptuales

Área de conocimiento	Principio conceptual
KA1. Requerimientos del software	La definición de la función intencional es fundamental para asegurar que el producto software representa un valor agregado para el negocio. Para tal propósito, se hace necesario conocer, seleccionar y aplicar prácticas de captura, análisis y gestión de requisitos que faciliten un esquema permanente de interacción con el cliente y una evolución de las necesidades para convertirse en especificaciones que debe cumplir el producto software (Kotonya y Sommerville, 1998; Schneider y Winters, 1998).
KA2. Diseño de Software	La complejidad creciente del software imponen en el diseño dos connotaciones diferentes: a un alto nivel, el diseño representa la estructura en módulos o componentes con funcionalidad claramente establecida y su interrelación entre dichos partes, la cual se denomina *arquitectura* y a un nivel detallado que especifica la estructura y comportamiento interno de cada uno de módulos, la cual se denomina *micro arquitectura*. Estas dos perspectivas de la estructura del sistema permiten un manejo adecuado de la complejidad, que facilitan que el desarrollador se concentre primero en decisiones de alto nivel que logran acuerdos entre los involucrados en el proyecto, y la decisiones del diseño detallado, que se encuentran cercanas a consideraciones de implementación del producto (Bass, 2003). UML como lenguaje estándar que puede ser utilizado tanto a nivel de arquitectura como de micro arquitectura. El desarrollo de la tecnología alrededor de XML como lenguaje estándar de intercambio y de XMI como lenguaje estándar de representación e intercambio de modelos UML, ha generado una propuesta conceptual de arquitectura, promovida por la OMG, llamada MDA[1], la cual propone mecanismos y lenguajes para abordar el proceso de desarrollo de software alrededor de los modelos (Kleppe, *et.al.* 2004).
KA3. Construcción del software	En la construcción deben considerarse los siguientes principios: minimizar complejidad, anticipación al cambio, construcción para verificación y seguimiento de estándares. Es importante hacer una adecuada planificación de la construcción y mantener mediciones acerca del desempeño y calidad de los programadores. Se deben conocer y manejar con propiedad los diferentes lenguajes que apoyan el proceso de construcción en los diferentes niveles de la arquitectura: lenguajes de presentación, lenguajes de programación, lenguajes de configuración, lenguajes de bases de datos (SWEBOK, 2004).
KA4. Pruebas de software	Las pruebas de software consisten en la verificación dinámica del comportamiento de un programa en un conjunto finito de casos de prueba, adecuadamente seleccionado de los posibles escenarios del sistema, para asegurarse que arroja el resultado definido en la especificación (SWEBOK, 2004). Los casos de prueba surgen desde la fase de análisis de requisitos. Importancia de desarrollar una disciplina de pruebas unitarias por parte de los programadores. Importancia de herramientas que automaticen los diferentes tipos de prueba.

[1] Model Driven Architecture

Área de conocimiento	Principio conceptual
KA5. Mantenimiento del software	El mantenimiento de software es una disciplina que ha recibido poca atención como parte identificable dentro del ciclo de vida. Aunque se puede utilizar buena parte de las técnicas utilizadas en el análisis y diseño, es importante analizar los escenarios particulares en los que ocurre el mantenimiento (como interfaces con otros sistemas, migración de aplicaciones legado, conversión de programas a nuevas plataformas de hardware y software) y apropiar técnicas propias de esta tarea (reingeniería e ingeniería inversa) (SWEBOK,2004).
KA6. Gestión de configuración	Es la disciplina que permite la identificación de la configuración del sistema en diferentes puntos del tiempo. La implementación de esta disciplina parte de unos lineamientos administrativos para identificar los artefactos (ítems) que van a entrar al control de configuración y técnicos para seleccionar las herramientas de apoyo al control de versiones (SWEBOK, 2004).
KA7. Gestión de la ingeniería de software	El modelo de ciclo de vida debe ser seleccionado de acuerdo a las condiciones propias de un proyecto particular. Modelos como el incremental y el espiral, favorecen la construcción progresiva del producto software (Lawrence, 2002; Pressman, 2006). Las prácticas ágiles están orientadas hacia del diseño detallado y la programación, y proponen esquemas de trabajo dinámicos y responsables entre los miembros del equipo de trabajo (Beck y Fowler, 2000). La naturaleza del proceso software como actividad humana, hace necesario que se comprenda el papel que juega este recurso (individuo, equipo, cliente, organización) en el proceso de desarrollo, y las variadas y complejas interacciones que se suceden entre las personas, que desde diferentes perspectivas contribuyen, ya sea a impulsar el avance del proyecto o a propiciar su fracaso (Yu y Mylopoulos, 1994; Humphrey, 2000, Tomayko, 2004). El SEI reconoce la importancia de acompañar las propuestas de madurez del proceso, con guías que permiten la debida inserción del recurso humano en tales procesos, tanto de manera individual a través de los enfoques dados en el modelo PSP[2] (Humphrey, 1997) como de manera colectiva a través del modelo TSP [3] (Humphrey, 2000). Las propuestas de desarrollo extremas, como XP (Beck, 2000), conceden especial importancia al recurso humano como actor principal alrededor del cual se garantiza la calidad del producto.
KA8. El proceso de ingeniería de software	Esta área analiza el proceso, como marco de trabajo que establece lineamientos a nivel organizacional, denominado modelo de procesos. Existen modelos del proceso software que son referentes a nivel mundial, como CMM[4], CMMI[5], BOOTSTRAP, ISO/IEC TR 15504 (SPICE) (Wang y King, 2000). Existen otros modelos, como Moprosoft (Oktaba, 2003) y Métrica 3 (Métrica 3), definidos como iniciativas nacionales, con el propósito de articularlos a las culturas y realidades del contexto sin los altos costos que implica la adopción de modelos internacionales como los referidos en el primer grupo.

2 Personal Software Process

3 Team Software Process

4 Capability Maturity Model

5 Capability Maturity Model Integration

Área de conocimiento	Principio conceptual
KA9. Métodos y herramientas de ingeniería de software	Un método de desarrollo de software está directamente influenciado por los principios de ingeniería[6], de los que se parte para enfrentar la complejidad inherente al problema con el propósito de descomponerlo en unidades manejables (Meyer, 1999). Dichos principios dan origen a las principales aproximaciones de desarrollo de software como OO (Jacobson et.al, 1999), CBSE (Szyperski, 1998; Cheesman y Daniels, 2001), AOSD (Rashid *et. al.*, 2003; Clarke, 2004). Existe una diversidad de herramientas que apoyan el proceso de ingeniería en sus diferentes fases como Herramientas CASE que soportan el modelado de las aplicaciones y la generación de código[7], entornos integrados de desarrollo (IDE) que agilizan la construcción de código, manejadores de versiones que proveen mecanismos para el control y evolución de los artefactos dentro de un equipo de trabajo, herramientas de pruebas y herramientas para definición, gestión y trazabilidad de los requisitos. Se requieren además herramientas que soporten los procesos de gestión y apoyo, herramientas especializadas para gestionar el proyecto de desarrollo software, herramientas de gestión de incidentes y herramientas para medir la calidad de los artefactos. Tanto lo métodos como las herramientas de apoyo se deben articular a la respectiva disciplina. Un mismo método puede servir a mas de una disciplina.
KA10. Calidad del software	Los primeros modelos de calidad estaban orientados a evaluar la calidad del producto final; hoy se reconoce la importancia de establecer mecanismos para evaluar la calidad del producto en sus fases intermedias. La revisión técnica formal (RTF) es una de las técnicas más utilizadas para realizar la revisión o inspección de un producto. Es posible, además, establecer modelos que cuantitativamente determinen el nivel de calidad de los diversos artefactos generados a lo largo del proceso de desarrollo (Género *et. al.,* 2004). Existen estándares internacionales como el ISO/IEC 9126 (ISO9126), que establecen las características que deben ser evaluadas en un producto software. El énfasis en la arquitectura ha favorecido el análisis de calidad del producto en las fases iniciales. Se proponen guías y procedimientos que permiten realizar una evaluación cuantitativa de la arquitectura y la manera como ésta da cumplimiento a atributos de calidad (Clements, *et.al.* 2002).

2. Referente de Contexto: la práctica del desarrollo en las empresas de software

Este análisis representa una visión del estado de la práctica en las empresas de software; se tienen en cuenta aquellos aspectos en los que se considera que la academia puede tener ingerencia, con el propósito de contribuir a mejorar la práctica del desarrollo de software en dichas organizaciones.

Una de las acciones encaminadas a mejorar tal práctica, es la adopción de modelos de calidad. Desafortunadamente la adopción de un modelo de este tipo surge como una decisión gerencial motivada principalmente por la certificación como un fin en sí mismo, no como un esfuerzo integrado de todos los involucrados con el objetivo de mejorar la cadena productiva. Esta situación genera problemas como los siguientes:

[6] Abstracción, ocultamiento de la información, descomposición funcional, modularidad, reutilización, separación de asuntos.
[7] En la mayoría de las herramientas CASE dicha generación se realiza de manera parcial.

Problema 1. Desarticulación entre los enfoques de calidad a nivel organizacional y las necesidades de mejoramiento a nivel del proyecto. El enfoque de calidad puede definirse como la estrategia con la que la organización emprende el trabajo de adopción de un modelo de calidad. Generalmente los esfuerzos se concentran en la definición de procesos más orientados hacia la producción de documentos que hacia la definición de la arquitectura a través de modelos; los desarrolladores perciben el modelo de calidad como una imposición de diligenciamiento de formatos que los distrae del trabajo productivo y que no agrega valor a su trabajo diario.

Problema 2. Los procesos definidos son rígidos y no se adaptan al tipo de proyecto específico. En algunas compañías, el modelo de calidad representa una propuesta rígida de procesos, indicadores y documentos, que puede resultar pesada cuando se trata de proyectos pequeños o de complejidad baja.

Problema 3. Falta aplicación de buenas prácticas. En algunas compañías no se aplican prácticas que apoyen las tareas de ingeniería (requisitos, análisis, diseño y construcción, etc.) o las tareas de soporte a la gestión de calidad (manejo de configuraciones, verificación, etc.). En algunos casos, la aplicación de buenas prácticas se realiza de manera individual, de acuerdo con las habilidades del desarrollador, quedando en manos del protagonista de turno, el éxito o fracaso de un proyecto.

Problema 4. La adopción de UML, más para documentar que para especificar. Buena parte de las empresas aún utilizan UML como "lenguaje de documentación" debido a las exigencias del cliente, sin entender que el uso adecuado de este lenguaje obliga una nueva forma de analizar un problema y definir una solución siguiendo los principios de orientación por objetos. Estas compañías continúan construyendo las aplicaciones al estilo estructurado, concentrando la mayor parte de la lógica en lenguajes tipo *script* (jsp, asp) o en objetos de control que directamente acceden las estructuras de bases de datos.

Problema 5. El ciclo de vida que se sigue no se ajusta al tipo de proyecto. Algunas empresas de software siguen empleado un ciclo de vida en cascada, que ofrece poca flexibilidad para ir construyendo progresivamente la solución y que no favorece la interacción con los usuarios a lo largo del proyecto. En otras empresas se percibe una inadecuada adopción del modelo XP, más como manera de justificar el estilo un tanto desorganizado de construir una solución software, que como aplicación de prácticas formales de diseño y programación ampliamente soportadas en herramientas. Aunque el modelo XP puede resultar adecuado para aplicaciones de tamaño pequeño, vale la pena reconsiderar su adopción en aplicaciones complejas de gran tamaño, con buen número de usuario involucrados, en las que la definición de la arquitectura, como acuerdo de diseño de los participantes del proyecto, es de vital importancia para garantizar una evolución controlada de la solución y favorecer la construcción de componentes o servicios genéricos que pueden ser reutilizados.

Problema 6. Dificultad para adoptar nuevas aproximaciones de desarrollo. En general, se percibe en los desarrolladores una resistencia al cambio ante nuevos aproximaciones de desarrollo propuestas desde la ingeniería de software. Esta resistencia puede obedecer a la exigencia en el cumplimiento de procesos rígidos que no se adaptan a la naturaleza particular del proyecto (problema #2) o a la debilidad en la competencia del desarrollador para adaptarse a ellas.

Problema 7. Gestión inadecuada en desarrollo basado en iteraciones. Algunas empresas han adoptado una gestión del proyecto basada en iteraciones. Se percibe que la construcción de módulos se inicia en fases muy tempranas del proyecto, sin haber realizado una definición de la arquitectura que represente el acuerdo de todos los involucrados. En algunas situaciones no se hace una adecuada planificación de los casos de uso a ser implementados en cada iteración: éstos se desarrollan según la prioridad que establece el cliente sin considerar el impacto de la funcionalidad en la arquitectura del sistema.

Problema 8. Mayor énfasis en pruebas finales que en revisiones intermedias. Buena parte de las empresas orientan el esfuerzo de verificación de

la calidad sobre el producto final, haciendo poco énfasis en la revisión de los artefactos generados en las etapas intermedias, incurriendo en entrega de productos defectuosos y en altos costos de corrección de errores.

Problema 9. Falta de métricas que midan la calidad en productos intermedios. Algunas empresas reconocen la importancia de realizar procesos de revisiones formales de los artefactos software en las etapas iniciales del desarrollo. Aunque algunas de ellas cuentan con listas de chequeo estandarizadas para un artefacto en particular, no se tienen métricas que permitan evaluar cualitativamente la calidad del producto y que se refleje en indicadores de productividad y calidad en las diferentes fases del desarrollo.

Problema 10. Elaboración parcial de la arquitectura. En buena parte de los proyectos de desarrollo, la concepción de la arquitectura como acuerdo en las fases tempranas gira alrededor de la infraestructura tecnológica (hardware y software de base) y de las decisiones relacionadas con la distribución y despliegue del producto, dejando de lado elementos que son el núcleo de una solución de arquitectura: la definición de los componentes de alto nivel y las interrelaciones entre éstos.

Problema 11. Falta de claridad en el rol del arquitecto. Se percibe la dificultad de establecer diferencia entre el diseño de alto nivel y el diseño detallado; esta dificultad se evidencia en que el arquitecto de la solución se convierte en muchos casos en el diseñador detallado del sistema, perdiendo de esta manera la visión global que se desea que conserve, con el propósito de apropiar y definir estándares y patrones de desarrollo que apliquen a una familia de sistemas.

Problema 12. Herramientas CASE como apoyo a la documentación antes que a la especificación. Buena parte de los desarrolladores utilizan el CASE como herramienta de documentación de los modelos una vez la solución está construida, lo cual refleja el problema de fondo: no se tiene un proceso de desarrollo soportado en la construcción de modelos y, por lo tanto, difícilmente estará orientado a la arquitectura.

Problema 13. Falta de herramientas automatizadas e integradas. Se percibe una falta de utilización de herramientas que apoyen el proceso de desarrollo en sus diferentes actividades; existen dos problemas básicos: falta de integración de las herramientas y alto costo de las soluciones integradas. Existe una interesante tendencia al uso e integración de herramientas de código abierto que soportan las tareas del desarrollador de software.

3. Principios y estrategias que soportan la enseñanza de la Ingeniería de Software

Hasta este momento hemos analizado el estado del conocimiento en el desarrollo de software y el estado actual de la práctica del desarrollo en las empresas de software. Estos dos elementos sirven como referentes conceptuales y de contexto, con el propósito de definir los principios y las estrategias en los que se debe fundamentar la enseñanza de la ingeniería de software. Dichos principios y estrategias estarán fundamentados en teorías alrededor del proceso de aprendizaje.

La pedagogía y la didáctica proveen elementos para asumir las relaciones complejas entre el instructor y el aprendiz, respecto al quehacer cognitivo alrededor de un objeto de estudio, y el papel que juegan los diferentes instrumentos incorporados al proceso. Se toma como referencia la propuesta pedagógica de la teoría constructivista del aprendizaje, definida por Bruner (Bruner, 2001), quien describe el proceso de aprender, los distintos modos de representación y las características de una teoría de la instrucción. Partiendo de los postulados pedagógicos, definiremos los principios aplicados al área de conocimiento de nuestro interés y formularemos las estrategias que deben ser tenidas en cuenta dentro del proceso de enseñanza de la ingeniería de software.

3.1 Postulado Pedagógico: El aprendizaje como proceso de categorización

El aprendizaje consiste esencialmente en la categorización de las cosas con el propósito de simplificar la interacción con la realidad y facilitar la acción (Bruner, 2001). La categorización está estrechamente relacionada con procesos como la selección

de información, generación de proposiciones, simplificación, toma de decisiones y construcción y verificación de hipótesis. El aprendiz interactúa con la realidad organizando las entradas según sus propias categorías, posiblemente creando nuevas o modificando las preexistentes. El aprendizaje es por lo tanto un proceso activo, de asociación y construcción.

Principio 1. El desarrollo de software como un proceso de abstracción. La principal competencia que se busca desarrollar en el estudiante es su capacidad de abstracción. El estudiante se acerca progresivamente a un problema para entenderlo, analizarlo, representarlo y validarlo con el propósito de construir una solución software. La abstracción trae implícita la habilidad de destacar aquellos elementos de la realidad que son relevantes a un determinado nivel de abstracción. El principio de abstracción como fundamento del proceso de desarrollo de software, está directamente soportado en la categorización, como proceso mental que sigue un aprendiz para entender y transformar una realidad.

Estrategia 1.1. Teniendo en cuenta el nivel de madurez del estudiante y el conocimiento que en un semestre determinado ha adquirido desde las diferentes áreas de formación, se propone una estrategia de desarrollo de capacidades de abstracción que va desde las más concretas como diseñador hasta las más abstractas como analista de requisitos, arquitecto y, finalmente, gerente de proyecto.

Estrategia 1.2. Las técnicas utilizadas para representar un problema o su solución deben estar orientadas, primeramente, a encontrar los conceptos relevantes del problema para, luego, caracterizarlos.

Estrategia 1.3. Resaltar la importancia de la definición de la arquitectura como diseño de alto nivel, que permite enfrentar la complejidad de una solución.

3.2 Postulado pedagógico: El grado de asimilación de un nuevo conocimiento está directamente relacionado con la estructura cognitiva previa del aprendiz

La estructura mental del aprendiz da significación y organización a sus experiencias y le permite ir más allá de la información. El enfoque constructivista visualiza el aprendizaje como un proceso en espiral, en el que se van trabajando periódicamente los mismos contenidos, cada vez con mayor profundidad, con el fin de que el estudiante continuamente modifique las representaciones mentales que ha venido construyendo.

Principio 2. Formación iterativa e incremental de los elementos involucrados en el desarrollo de software. Repetición y redundancia de conceptos clave del desarrollo que van siendo refinados de un curso a otro. Se parte de una visión general de la ingeniería de software, que va siendo precisada desde diferentes dimensiones a medida que se avanza en el ciclo de formación.

Estrategia 2.1. Lograr una visión común de los profesores de todas las asignaturas que tienen involucrado el desarrollo de software, de tal manera que existan algunas formas de trabajo común. Disponer de un glosario de términos alrededor de los conceptos clave, que sea referente común a todos los estudiantes y profesores del área. El uso de prácticas, como diseño y programación por contratos (Meyer, 1999), pruebas unitarias, RTF, seguimiento de estándares, debe ser difundido y promocionado en todas las asignaturas relacionadas.

Estrategia 2.2. Enfocar el proceso de formación alrededor de los elementos básicos de la ingeniería de software: el proceso, el producto, los métodos y técnicas, la tecnología y el proyecto. Esto significa que la estructura de las materias del área deben hacer explícita la manera como trabajan las diferentes dimensiones de formación.

Estrategia 2.3 Adoptar un marco de referencia del proceso software como referente común a lo largo de todo el ciclo de formación. Dicho marco debe articular los principios, procesos estándares, métodos y buenas prácticas, que pueden ser aplicados al desarrollo de software. El estudiante debe adquirir la habilidad de analizar con sentido crítico la manera como el proceso, los métodos y las herramientas deben ser aplicados.

3.3 Postulado pedagógico: Autoregulación del aprendizaje

Es importante que el estudiante reflexione en todo momento acerca de los procesos mentales que sigue para asumir, acomodar y transferir un nuevo concepto o producto, modificando sus estructuras mentales y de comportamiento, que se articulan a las experiencias que ya tiene (Blythe, 1998).

Principio 3. Conciencia permanente de la calidad. La calidad vista como apropiación de buenas prácticas de ingeniería, alrededor tanto del producto que se construye como del proceso que se sigue. En los primeros cursos se enfatiza en la formación de hábitos que, de manera inconsciente, vayan favoreciendo en los estudiantes la apropiación natural de buenas prácticas que luego serán formalizadas desde la teoría misma.

Estrategia 3.1. Propiciar que en cada trabajo práctico el estudiante pueda reflexionar acerca del proceso que siguió, la técnica que utilizó y las herramientas que aplicó.

Estrategia 3.2. Concientizar desde los primeros cursos de programación sobre la importancia de las pruebas y favorecer el desarrollo de habilidades en el manejo de herramientas de pruebas (Shepard, 2001; Edwards, 2003).

Estrategia 3.3. Favorecer desde los primeros cursos actividades de revisión por pares. Los estudiantes deben aprender a aceptar críticas pro-activas de sus compañeros, que le ayudan a mejorar los diferentes procesos (Braken, 2003).

Estrategia 3.4. Enfatizar desde los primeros semestres la importancia de la calidad del software, tanto interna como externa. La calidad interna, por medio de características evaluables como cohesión y acoplamiento (Meyer, 1999), y la calidad externa a través de características como el rendimiento (Dugan, 2004).

Estrategia 3.5. Enfatizar desde los inicios en la reutilización, por medio del adecuado uso de librerías de programas desarrollados por terceros; promover la creación de librerías / componentes que puedan ser utilizadas de un semestre a otro.

Estrategia 3.6. Proponer instrumentos de evaluación que permitan al profesor y a los estudiantes reflexionar acerca de la efectividad del proceso de formación en cada uno de los cursos (Stevens, 2001).

Estrategia 3.7. Propiciar desde la teoría y la práctica una reflexión permanente acerca del modelo de ciclo de vida, que debe ser aplicado dependiendo del tipo de proyecto, y de las implicaciones de cada uno de estos modelos.

Estrategia 3.8. Favorecer en los trabajos prácticos el uso de herramientas de apoyo a todo el proceso de desarrollo en todas sus modalidades: de modelado, de gestión, de calidad.

3.4 Postulado Pedagógico: El desequilibrio cognitivo es clave para lograr la asimilación y acomodación de nuevos conceptos

Este principio tiene como propósito enfrentar al aprendiz a un proceso permanente de asimilación y acomodación que lo obliga a construir nuevos modelos que amplíen su estructura intelectual.

Principio 4. Permanente acercamiento a la realidad. En cada una de las asignaturas se debe favorecer una aplicación del conocimiento en casos de estudio suficientemente representativos o casos reales que permitan al estudiante el ajuste y articulación de sus conocimientos a una situación concreta, con el propósito de modificarla. Este acercamiento favorece, además, que el estudiante perciba la necesidad de apropiarse de nuevos métodos o herramientas.

Estrategias 4.1. Favorecer acercamiento a tecnologías actuales.

Estrategia 4.2. Favorecer esquemas de trabajo en los que los estudiantes de los primeros semestres puedan participar en proyectos reales o de otros cursos.

Estrategia 4.3. Favorecer esquemas de trabajo en los que los estudiantes tomen parte activa en proyectos desarrollados dentro de las empresas de software.

3.5 Postulado Pedagógico: La representación como reflejo del desarrollo cognitivo del estudiante

En su modelo pedagógico, Bruner enfatiza la capacidad de representación en contexto, como el reflejo del desarrollo cognitivo del estudiante. En el área del desarrollo de software, la capacidad de representación es una de las competencias básicas que permiten al estudiante ajustarse sin mayores problemas a nuevas propuestas de lenguajes, herramientas y notaciones.

Principio 5. Independencia de la semántica y la notación. El uso de un método para entender un problema y modelar una propuesta de solución, debe ser una competencia inherente del desarrollador de software. En cada acercamiento progresivo al problema y a su solución, el estudiante debe estar capacitado para seleccionar el método más adecuado para su representación, de acuerdo con la semántica que se intenta capturar. Esto significa que él debe adquirir la habilidad de independizar la semántica del problema, de la notación o técnica que se utilice; mientras que la primera provee los principios fundamentales para entender y modelar el problema, la segunda provee las herramientas para su representación.

Estrategia 5.1. Proveer en los cursos de programación una visión general de las diferentes aproximaciones de programación, de forma que, en las asignaturas de ingeniería de software, se puedan abordar como aproximaciones para el desarrollo de software. Enfrentar permanentemente al estudiante con los principios en los que se fundamentan las diversas aproximaciones de desarrollo y evaluar las nuevas aproximaciones a la luz de dichos principios.

Estrategia 5.2. Utilizar al menos un lenguaje de referencia común, como UML, para especificación de soluciones software. No obstante, los estudiantes deben estar capacitados para entender y apropiar otros lenguajes de representación, a la luz del nivel de abstracción en el que se encuentre y el propósito particular que se desee alcanzar.

Estrategia 5.3. Enfatizar a través de los cursos la importancia de modelos independientes de la plataforma en la que va a ser implementado y que luego pueda hacerse corresponder a una plataforma particular

3.6 Postulado pedagógico: La conducta humana como reflejo integrado de lo cognitivo y lo afectivo

La conducta humana, entendida como la manifestación del proceso de adaptación del sujeto a la realidad, va más allá del aspecto estructural y cognitivo, influenciando sus sentimientos y afectos. El objetivo final del proceso de aprendizaje habrá sido alcanzado si se logra que los sentimientos que el aprendiz experimenta ante el descubrimiento del conocimiento, se consolidan en propósitos que norman su voluntad y canalizan la energía afectiva, de tal manera que provocan interés, motivación y satisfacciones internas. A través del proceso de aprendizaje, se espera que el estudiante desarrolle valores como la organización disciplinada de tiempos, la autocrítica y la valoración de los procesos auto formativos..

3.7 Postulado pedagógico: La razón como producto de interacción social

La interacción social ejerce sobre los sujetos relaciones de presión y de colaboración. A través de la confrontación con otros, el sujeto analiza su propio pensamiento y toma conciencia de su modo de organización, de tal manera que enriquece el mecanismo de la razón..

Principio 6. Trabajo personal vs. trabajo en equipo. Los dos postulados anteriores permiten fundamentar la importancia, tanto del trabajo individual como del trabajo en equipo, como instrumento de formación integral del aprendiz. Mientras que el primero tiene por objetivo asegurar que el estudiante interiorice adecuadamente su experiencia de formación, el segundo tiene el propósito de capacitar el estudiante como miembro de un equipo de trabajo, con manejo adecuado de interacciones tanto de presión como de colaboración.

Estrategia 6.1. Las asignaturas deben propiciar un adecuado equilibrio entre actividades que enfaticen

tanto el trabajo individual como el trabajo en equipo.

Estrategia 6.2. A través de los trabajos prácticos los estudiantes deben adquirir una clara conciencia de los diferentes roles involucrados en el desarrollo de software. Especial atención merece el rol de arquitecto de software como director técnico que lidera el acuerdo de diseño de alto nivel, teniendo en cuenta las consideraciones del negocio y las características de la infraestructura tecnológica.

Conclusiones

Los docentes y las instituciones educativas desempeñan un papel relevante para lograr que los desarrolladores de software sean personas comprometidas, cuya competencia natural sea desarrollar sistemas software de calidad. Este artículo ha presentado una visión general de la enseñanza de la ingeniería de software, que parte del estado actual del área de conocimiento como referente conceptual y de los problemas vigentes en las empresas de software alrededor de la práctica de desarrollo, como referente contextual. Se formulan los principios y estrategias en los que se debe soportar la enseñanza de la ingeniería de software, los cuales están fundamentados en postulados formulados en el campo de la pedagogía y la didáctica.

El modelo SWEBOK es el cuerpo de conocimiento que puede ser utilizado como referente conceptual a la hora de definir el modelo de enseñanza de la ingeniería de software tanto a nivel de pregrado como de postgrado. El proceso, el producto, el método, las personas y la tecnología, con todas sus variantes, son elementos que están siempre presentes en la práctica del desarrollo de software y que deben ser articulados de forma adecuada en la estrategia pedagógica. El desafío de la academia es propiciar espacios para que, desde lo teórico y lo práctico, el estudiante desarrollo criterios que le permitan intervenir la realidad del desarrollo de software en las organizaciones, con el propósito de modificarla.

La formación de competencias y habilidades orientadas a la aplicación de buenas prácticas, debe realizarse a lo largo de todo el ciclo de formación siguiendo un proceso iterativo e incremental. Para lograr tal propósito es necesario que los docentes relacionados con esta disciplina mantengan una reflexión permanente, tanto de los elementos conceptuales que dan soporte al área, como de los problemas que existen con respecto a la aplicación de dichas prácticas en el contexto real. De esta manera se logrará en el estudiante una formación progresiva de competencias a lo largo de su ciclo de formación, que sea pertinente para las demandas del medio empresarial.

Los principios y estrategias presentados, surgen como resultado de la experiencia en la enseñanza de la ingeniería de software, de la reflexión del área de conocimiento desde el campo investigativo y de las experiencias de participación cercana en proyectos reales e interacciones con empresas de software. Las estrategias aquí presentadas representan líneas de acción que contribuirán al desarrollo de competencias de los estudiantes, con el propósito de que éstos se conviertan en promotores de cambio en las organizaciones de software. Por razones de espacio, nos limitaremos a realizar en la Tabla 2 una síntesis de la manera como las estrategias pedagógicas contribuirán a subsanar los problemas detectados en la práctica del desarrollo de software a través de la formación de competencias específicas en el área de la ingeniería de software.

Tabla 2. Contribución de las estrategias pedagógicas en la solución de los problemas en la práctica del desarrollo de Software

Problema detectado	Estrategias asociadas	Competencia a desarrollar
Problema 1. Desarticulación entre calidad organizacional y necesidades del proyecto	3.1 a 3.6	Involucrar de forma natural, en todas las actividades del desarrollo de software, prácticas de aseguramiento de calidad.
Problema 2. Rigidez de los procesos definidos	2.3	Poseer claridad conceptual frente a la variabilidad del proceso ante condiciones particulares de un proyecto.
Problema 3. Falta aplicación de buenas prácticas	2.3	Aplicar con propiedad prácticas reconocidas que apoyen las tareas de definición de requisitos, análisis, diseño y construcción.
Problema 4. La adopción de UML para documentar más que para especificar	5.1 a 5.3	Capacidad de modelar como estrategia para entender un problema y especificar su solución.
Problema 5. Ciclo de vida que no se ajusta al tipo de proyecto	3.7 y 4.3	Reconocer el impacto de la selección del ciclo de vida dependiendo de las condiciones de contexto de un proyecto.
Problema 6. Dificultad para adoptar nuevas aproximaciones de desarrollo	5.1 a 5.3	Entender las semejanzas y diferencias de los procesos, métodos y lenguajes que apoyan el desarrollo de software, contando con criterios para decidir su aplicación en un contexto particular.
Problema 7. Gestión inadecuada en desarrollo basado en iteraciones	3.7 y 4.3	Aplicar de forma adecuada los principios de gestión del proceso iterativo de desarrollo.
Problema 8. Mayor énfasis en pruebas finales que en revisiones intermedias.	3.3 y 3.4	Aplicar de forma natural procesos de validación y revisión temprana de la arquitectura.
Problema 9. Falta de métricas que midan la calidad en productos intermedios	3.3 y 3.4	Aplicar mediciones que evalúan el cumplimiento de atributos de calidad de un producto software.
Problema 10. Elaboración parcial de la arquitectura	1.3 y 6.2	Reconocer la importancia de la arquitectura como acuerdo
Problema 11. Falta de claridad en el rol del arquitecto	6.2	Tener clara conciencia de los roles involucrados en un proyecto de desarrollo.
Problema 12. Herramientas CASE como apoyo a la documentación, antes que a la especificación	3.8, 5.1 a 5.3	Manejar herramientas CASE como apoyo a la actividad de modelado.
Problema 13. Falta de herramientas automatizadas e integradas	3.8	Manejar herramientas (preferiblemente de código abierto) que pueden apoyar diversas tareas de desarrollo (pruebas, manejo de versiones, trazabilidad, etc.)

Actualmente se está trabajando en la definición de un modelo para la enseñanza de la ingeniería de software enfocado hacia el desarrollo de competencias en los estudiantes de los ciclos de formación básica y de postgrado. La definición del perfil en términos de competencias es una tendencia que se sigue actualmente con el propósito de lograr en el egresado una combinación de actitudes, estructuras mentales, habilidades y destrezas, a fin de que pueda desempeñar determinadas acciones profesionales una vez finalizado un programa formativo (López, 2003; Letelier et.al, 2005). El reto en la definición de dicho modelo, es lograr una articulación completa de las unidades temáticas y las actividades de aprendizaje partiendo de las estrategias aquí planteadas. Mientras que los referentes conceptuales estarán soportados en SWEBOK, las actividades de aprendizaje estarán orientadas al desarrollo de competencias que capacitan al estudiantes para la aplicación de prácticas reconocidas en modelos como CMMI (Crisis, 2003).

Bibliografía

Bass, Less *et. al.* (2003). S*oftware architecture in practice*. 2. ed. Addison Wesley.

Beck, Kent y Fowler, Martin. (2000). *Planning extreme programming.* Addison Wesley Professional.

Blythe, Tina (1998). *Enseñanza para la comprensión: guía para el docente.* Editorial Paidos.

Bracken, Barbara (2003). "Progressing from student to professional: the importance and challenges of teaching software engineering". En: *Journal of Computing Sciences in Colleges***.** Dec. Vol. 19. No. 2.

Bruegge Bernd y Dutoit, Allen (2002). *Ingenieria de software orientado a objetos.* Prentice Hall.

Bruner, Jerome (2001). *Realidad mental y mundos posibles.* Editorial Gedisa.

Cheesman, John; Daniels, John. (2001). *UML Components.* Addison Wesley.

Clements Paul, *et.al.* (2002). *Evaluating software architectures. Methods and case studies.* Addison Wesley.

Crisis, Mary, *et.al.* (2003). *CMMI. Guidelines for Process Integration and Product Improvement.* Addison Wesley.

Dugan, Robert (2004). "Performance lies my professor told me: The case for teaching software performance engineering to undergraduates". En: *Workshop on Software and Performance Proceedings of the fourth international workshop on Software and performance.* California: Redwood Shores.

Edwards, Stephen (2003). "Improving Student Performance by Evaluating How Well Students Test Their Own Programs, Virginia Tech". En: *ACM Journal of Educational Resources in Computing.* Vol. 3. No. 3.

Género, M.; Piattini, M.; Cruz, J.; Reynoso, L. (2004). En: "Metrics for UML models". *The european Journal for the Informatic Professional. Vol. V. No. 2.*

Greenfield, J. y Short, K. (2004). *Software Factories Assembling Applications with Patterns, Models, Frameworks, and Tools.* Wiley Publishing.

Humphrey, W.S (1997). *Introduction to the Personal Software Process.* Reading, MA: Addison–Wesley.

Humphrey, W.S (2000). *Introduction to team software process.* Reading, MA: Addison-Wesley.

ISO9126 (2003). *Sofware engineering product quality. Part 1,2,3. ISO/IECTR 9126-1,2,3.* Editorial Geneva. ISO/IEC

Jacobson, Ivar; Booch, Grady; Rumbaugh, James (1999). *El Proceso unificado de desarrollo de Software.* Addison Wesley.

Kleppe, Anneke (2004); Warner, Jos; Bast, Win. *MDA Explained. The model driven architecture: practice and promise.* Addison-Wesley.

Kotonya, G.; Sommerville, I. (1992). *Requirements engineering: processes and techniques.* John Wiley and Sons.

Kiczales, G. *et.al.* (2002). "Aspect-Oriented Programming". In: P*roceedings of ECOOP'97.* Dresden. Germany.

Kruchtem, P (1995). "The 4+1 View Model of Architecture". En*: IEEE Software.* Vol 12. No. 6.

Lawrence P., Shari (2002). *Ingeniería de software. Teoría y práctica.* Prentice Hall.

Liu, Jie; Marsaglia, John y Olson, David (2002). "Teaching software engineering to make studentes ready to realworld". En: *Journal of Computing Sciences in Colleges.* Vol. 17. No. 6.

Lieberherr, Karl J. (1996) *Adaptive Object-Oriented Software: the demeter method with propagation patterns.* Boston: PWS Publishing Company.

López P., Javier F. (2003). "Desarrollo de competencias en la formación de ingenieros de sistemas". En: *Revista ACIS*. Enero – Mayo de 2003. No. 84.

Letelier, M.; López, L.; Carrasco,R. y Pérez, (2005). "Sistema de competencias sustentables para el desarrollo profesional en ingeniería". En: *Revista Facultad de Ingeniería Universidad de Tarapacá.* Vol. 13. No. 2.

Mayer; Bunge (2004*) Panorama de laiIndustria del software en latinoamérica.* Brasil: Informática LTDA.

Maibaum, T.S.E (2004). *What We Teach Software Engineers in the University: Do We Take Engineering Seriously? One Academic('s) View of Software Engineering Education.* <From*:* www.cs.wm.edu/~coppit/ csci690-spring2004/papers>. (Consulta: Enero 2005)

Metrica 3. Ministerio de Administraciones Públicas, Metodología de Planificación, Desarrollo y Mantenimiento de sistemas de información. España. <From: http://www.csi.map.es/csi/metrica3/. (Consulta: Enero 2006).

Meyer, Betrand (1999). *Construcción de software orientado a objetos.* 2 ed. Prentice Hall.

Oktaba, Hanna *et.al* (2003). *Modelo de procesos para la industria de software. MOPROSOFT.* Versión 1.1.

Paulish, Daniel (2001). *Architecture centric software project management: A practical guide.* Addison Wesley.

Pressman, Roger (2006). *Ingeniería de software un enfoque práctico.* 6. ed. McGraw Hill.

Rashid, A.; Moreira, A. y Araújo, J. (2003). "Modularization and composition of aspectual Requirements". In: *AOSD.*

Schneider, G.; Winters (1998). *Applying use case.* Object Technology Series. Addison Wesley.

Shepard, Terry; Lamb, Margaret; Kelly, Diane (2001). "More testing should be taught". En: *ACM.* Vol. 44. No. 6.

Clarke, S.; Baniassad, E. (2004) *Aspect-Oriented analysis and design : The theme approach.* Object Technology Series. Addison-Wesley

Stevens, Tood (2001). *Experiences Teaching Software Engineering for the First Time.*

Department of Computer & Information Science University of Mississippi. <From: www.radford.edu/~kstevens2.> (Consulta: Enero 2005).

Stiller, Evelyn; LeBland Cathie (2002). "Effective software engineering pedagogy". En: *Journal of Computing Sciences in Colleges*. Vol. 18. No. 2.

SWEBOK (2004). *Guide to the Software Engineering Body of Knowledge. SWEBOK 2004* Version. A project of the IEEE Computer Society Professional Practices Comité.

Szyperski C. (1998). *Component software – beyond Object Oriented Programming*. Addison Wesley.

Tomayko, J.; Hazzan Orit (2004). *Human aspects of software engineering*. Charles River Media Computer Engineering Series.

Valdes, Luis E. (2004). "Situación Actual de la Informática en Colombia". En: *Informe del CATI*. Bogotá. Abril 2004.

Wang, Yingxu; King, Gram (2000). *Software engineering processes. Principles and application*. CRC Press.

Yu, Eric; Mylopoulos ,John (1994). "Understanding "why" in software process modelling, analysis, and design". In: *Proceedings of the 16th Int. Conf. Software Engineering*. Sorrento, Italy.